UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE ENGENHARIA DE COMPUTAÇÃO

PABLO MARTINS DA SILVA

Biblioteca de BDDs Baseada em Inteiros

Monografia apresentada como requisito parcial para a obtenção do grau de Bacharel em Engenharia da Computação

Orientador: Prof. Dr. André Inácio Reis

Porto Alegre
2019

“Binary decision diagrams (BDDs) are wonderful, and the more I play with them the more I love them. For fifteen months I’ve been like a child with a new toy, being able now to solve problems that I never imagined would be tractable.”

— DONALD E. KNUTH

AGRADECIMENTOS

Gostaria de agradecer a esta universidade, seu corpo docente, sua direção, seus administradores e todos os demais funcionários pela oportunidade de expandir meus conhecimentos e consequentemente poder ter um futuro melhor.

Também quero agradecer ao meu professor orientador, André Inácio Reis, pelo seu incentivo e pelas suas correções sem os quais este trabalho não seria possível.

Aos meus pais, Judite Martins e Vicente Lehmann da Silva, por me amarem desde o instante que nasci. Todo seu esforço, sua dedicação e seu apoio tornam minha vida muito melhor. Sem eles eu não estaria aqui.

Aos meus parentes, meus amigos e meus colegas que passaram pela minha vida por toda ajuda e por todos os ensinamentos. Em especial aos meus avós que apesar de não estarem mais presentes acredito que estão torcendo por mim. Outra menção especial é a minha amiga Clarissa Buarque que me ouve e ajuda da maneira que pode durante todos os dias.

Aos meus psicólogos e aos meus psiquiatras que me ajudaram a me reerguer nos momentos mais difíceis.

Por último e não menos importante, agradeço à minha cadela Yume cuja ausência deixa-me com saudade todos os dias. Obrigado por toda a companhia e momentos felizes que vivemos.

RESUMO

Diagramas de decisão binária (BDDs) são um tipo de estrutura de dados muito usada no projeto de circuitos integrados digitais. Este trabalho apresenta uma contribuição para a obtenção de estruturas de dados mais eficientes para BDDs através da proposta de uma chave única baseada em um só inteiro. Para operações lógicas de oito entradas ou mais, esta implementação foi de duas a treze vezes mais rápida do que uma implementação de referência onde a chave única é baseada em cadeias de caracteres. Para circuitos aritméticos cujas entradas possuem três bits ou mais, o resultado obtido foi um ganho de velocidade de 2,5 à 9,4 vezes.

Palavras-chave: BDDs. Circuitos Digitais. EDA. ROBDD.

ABSTRACT

BDDs are a type of data structure widely used in the design of digital integrated circuits. This work presents a contribution to obtain more efficient BDD data structures through using a unique key based on a single integer. For logic operations of eight inputs or more, this implementation was two to thirteen times faster compared to a reference implementation where the unique key is based in a string of characters. For arithmetic circuits of two three or more bits inputs, the result obtained was a 2.5 to 9.4 times gain in speed.

Keywords: BDDs, Digital Circuits, EDA, ROBDD.

LISTA DE ABREVIATURAS E SIGLAS

BDD	Binary Decision Diagram
CAD	Computer-Aided Design
CBDD	Chain Reduced Binary Decision Diagram
CZDD	Chain Reduced Zero-Supressed Binary Decision Diagram
EDA	Electronic Design Automation
EVBDD	Edge-Valued Decision Diagram
IBDD	Indexed Binary Decision Diagram
IDD	Interval Decision Diagram
ITE	If-Then-Else
LSB	Bit Menos Significativo
MDD	Multi-Valued Decision Diagram
MSB	Bit Mais Significativo
MTBDD	Multi-Terminal Binary Decision Diagram
OBDD	Ordered Binary Decision Diagram
ROBDD	Reduced Ordered Binary Decision Diagram
SCF	Strong Canonical Form
TBDD	Tagged Binary Decision Diagram
VLSI	Very-Large-Scale Integration
ZDD	Zero-Supressed Decision Diagram

LISTA DE FIGURAS

LISTA DE TABELAS

SUMÁRIO

1 INTRODUÇÃO

A criação dos circuitos integrados permitiu um grande avanço tecnológico para a humanidade, o que se reflete na melhoria da qualidade de vida. A constante taxa de diminuição das medidas dos transistores descrita por Gordon E. Moore, um dos fundadores da Intel, no que hoje é conhecida como Lei de Moore (MOORE et al., 1965) tem se mantido por várias décadas e permite que tenhamos equipamentos digitais cada vez mais sofisticados e poderosos. Porém, a complexidade do projeto desses sistemas também cresce na mesma medida. Por isso, o desenvolvimento de ferramentas de automação de projeto eletrônico (EDA, Electronic Design Automation, em inglês) cada vez mais eficazes contribui para a ubiquidade dos sistemas eletrônicos nos dias de hoje. A quantidade de transistores num processador pode chegar na casa dos bilhões e novas tecnologias estão sendo criadas para permitir que essa quantidade aumente ainda mais. Para lidar com projetos cada vez mais complexos e desafiadores, os projetistas utilizam técnicas tais como abstração e hierarquia. Isto envolve circuitos maiores tais como microprocessadores e microcontroladores (ROSA et al., 2003) trabalhados em altos níveis de abstração. Tais circuitos podem ser síncronos (ROSA et al., 2003) ou assíncronos (MOREIRA et al., 2014), mas sempre serão projetados em níveis de hierarquia onde os níveis mais baixos envolvem a representação de funções Booleanas. A manipulação destas funções Booleanas pode ser direcionada para tentar otimizar custos como area (MARTINS et al., 2010), atraso (JUNIOR et al., 2006) e potência (BUTZEN et al., 2010) (WILTGEN et al., 2013).

A representação e a manipulação eficiente de funções booleanas são importantes para diversos problemas nos softwares de EDA para projeto de sistemas digitais. Funções Booleanas podem ser representadas de várias formas, mas as três formas mais usadas são tabelas verdade representadas como inteiros, Grafos de Ands e Inversores (AIGs, do inglês And-Inverter-Graph) e Diagramas de Decisão Binários (BDDs, do Inglês Binary Decision Diagrams). Diferentes métodos podem ser baseados em diferentes estruturas de dados. O método de composição funcional usa tabelas verdade representadas como inteiros (MARTINS et al., 2010) (MARTINS; RIBAS; REIS, 2012). A biblioteca Kitty, uma das bibliotecas de síntese lógica da EPFL, é para a representação de funções Booleanas como inteiros (SOEKEN et al., 2018). O software ABC (MISHCHENKO et al., 2007), que é estado da arte em síntese lógica, é baseado principalmente em AIGs, embora todas as três estruturas sejam usadas internamente ao ABC, além de outras que não iremos discutir aqui, pois o ABC é um sistema cuja complexidade requer um entendimento

além dos requisitos de um Trabalho de Conclusão. BDDs podem ser usados em muitas aplicações, tais como gerar redes de transistores (JUNIOR et al., 2006) para células de biblioteca (TOGNI et al., 2002).

Este trabalho foca nos BDDs, que são uma ferramenta para resolver diversos tipos de problemas. Um desses problemas é a verificação lógica combinacional, que tem por objetivo verificar se um circuito combinacional gerado pela síntese lógica obedece a sua especificação (GEREZ, 1999). Knuth (2009) em seu livro fez um compilado de muitos problemas que podem ser resolvidos utilizando BDDs. O fato de Donald Knuth ter escrito um capitulo sobre BDDs é muito significativo para demonstrar a importância desta estrutura de dados.

1.1 Motivação

BDDs foram propostos por Lee (LEE, 1959) and Akers (AKERS, 1978). Porém só se popularizaram após Bryant (BRYANT, 1986) propor BDDs Reduzidos e Ordenados (ROBDDs) como uma estrutura de dados canônica para representação de funções Booleanas. A proposta da forma fortemente canônica dos ROBDDs popularizou ainda mais seu uso (BRACE; RUDELL; BRYANT, 1990). Nesta representação cada nodo de BDD tem uma chave única que é derivada da variável de controle do nodo e dos dois nodos filhos. Brace (BRACE; RUDELL; BRYANT, 1990) demonstrou que esta chave é única para uma mesma função Booleana e pode ser usada para a consulta da existência prévia do nodo, evitando a criação de nodos duplicados representando a mesma função Booleana.

A motivação deste trabalho é relativamente simples. A proposta é fazer uma implementação de BDDs onde a chave única que garante a canonicidade em ROBDDs na forma fortemente canônica é representada como um único inteiro. A implementação de referência é baseada em um pacote de BDDs que usa chaves canônicas baseadas em cadeias de caracteres. A expectativa inicial de que a chave única baseada em um inteiro apenas geraria uma implementação com tempo de execução mais eficiente foi plenamente atingida, resultando em uma implementação com tempos de execução 10 vezes mais rápidos.

1.2 Organização

O restante deste trabalho está organizado da seguinte forma. O capítulo 2 apresenta os conceitos necessários para entender este trabalho. O capítulo 3 descreve a implementação do pacote BDD baseado em inteiros. O capítulo 4 a metodologia utilizada para comparar a implementação deste trabalho com outra já existente e os resultados obtidos. Por fim, o capítulo 5 apresenta a conclusão deste trabalho de graduação.

2 FUNDAMENTAÇÃO TEÓRICA

Neste capítulo serão apresentados alguns conceitos necessários para o entendimento do deste trabalho. A seção 2.1 introduz a álgebra booleana e seus componentes para a compreensão das funções booleanas na seção 2.2. À seguir é apresentado o teorema da expansão de Shannon na seção 2.3 que é a base dos BDDs. A seção 2.4 mostra um resumo da história dos BDDs, seus conceitos fundamentais e um breve resumo sobre outras estruturas de dados derivadas dos BDDs.

2.1 Álgebra Booleana

Uma álgebra booleana é definida por uma 6-upla $(\mathbb{A}, +, \cdot, \neg, 0, 1)$, em que $\mathbb{A}$ é um conjunto; $+$ e $\cdot$ são operações binárias em $\mathbb{A}$; $\neg$ é uma operação unária em $\mathbb{A}$; e 0 e 1 são constantes em $\mathbb{A}$; que satisfaz os axiomas da tabela 2.1 para quaisquer elementos a, b ou c que pertençam a $\mathbb{A}$. Para fins gráficos a operação $\neg$ será substituída por uma barra superior desta forma, $\neg a = \overline{a}$. Alguns autores também incluem $0 \neq 1$ como um dos axiomas para evitar álgebras booleanas de um elemento, que são chamada triviais. Neste trabalho será considerada uma álgebra booleana de dois elementos.

A operação $+$ é chamada de conjunção, OU ou *OR*. A operação $\cdot$ se chama disjunção, E ou *AND*. A operação $\neg$ é conhecida como negação, complemento, NÃO ou *NOT*. As constantes 0 e 1 são também conhecidas como falso e verdadeiro, ou *false* e *true* respectivamente.

Tabela 2.1: Definição axiomática de uma álgebra booleana

Axioma	Forma Disjuntiva(+)	Forma Conjuntiva($\cdot$)
Comutatividade	$a + b = b + a$	$a \cdot b = b \cdot a$
Distributividade	$a + (b \cdot c) = (a + b) \cdot (a + c)$	$a \cdot (b + c) = (a \cdot b) + (a \cdot c)$
Elemento Neutro	$a + 0 = a$	$a \cdot 1 = a$
Complementar	$a + \overline{a} = 1$	$a \cdot \overline{a} = 0$

Fonte: Adaptada de Huntington (1904)

Esta definição axiomática da álgebra booleana atende aos critérios de correção e de completude. Como todas as fórmulas que podem ser deduzidas por esse conjunto de axiomas são válidas, então este sistema é correto. Como todas as fórmulas válidas podem ser provadas por este sistema, então ele é completo (HUNTINGTON, 1904). A tabela 2.1 apresenta uma lista de propriedades válidas que são úteis para realizar operações algébricas booleanas.

Tabela 2.2: Algumas propriedades da álgebra booleana

Propriedade	Forma Disjuntiva(+)	Forma Conjuntiva(·)
Associatividade	$(a+b)+c = a+(b+c)$	$(a \cdot b) \cdot c = a \cdot (b \cdot c)$
Idempotência	$a + a = a$	$a \cdot a = a$
Absorção	$a + (a \cdot b) = a$	$a \cdot (a+b) = a$
Elemento Absorvente	$a + 1 = 1$	$a \cdot 0 = 0$
Leis de De Morgan	$\overline{(a+b)} = \overline{a} \cdot \overline{b}$	$\overline{(a \cdot b)} = \overline{a} + \overline{b}$
Dupla Negação	$\overline{\overline{a}} = a$	

Fonte: Adaptada de Huntington (1904)

Além das operações apresentadas anteriormente, existem operações auxiliares que derivam delas e são muito utilizadas em circuitos digitais. A operação NÃO-E, também chamada de *NAND*, corresponde à negação da operação E, enquanto a operação NÃO-OU, que também é conhecida como *NOR*, é a negação da operação OU. Já as operações OU-EXCLUSIVO e NÂO-OU-EXCLUSIVO (*XOR* e *XNOR* respectivamente) tem relação com o número de entradas verdadeiras, a primeira é verdadeira se essa quantidade for ímpar e a segunda é verdadeira se essa quantidade for par. A tabela 2.1 apresenta os símbolos das operações XOR e XNOR, bem como sua expressão equivalente.

Tabela 2.3: Definição dos símbolos das novas operações

Operação	Símbolo	Expressão Equivalente
XOR	$a \oplus b$	$\overline{a} \cdot b + a \cdot \overline{b}$
XNOR	$a \otimes b$	$\overline{a} \cdot \overline{b} + a \cdot b$

Fonte: Os autores (2019)

2.2 Funções Booleanas

Uma função booleana é definida como uma função da forma $f : \mathbb{A}^n \to \mathbb{A}$ em que n é um número natural chamado de **aridade** da função. Caso $n = 0$, então a função é uma constante de $\mathbb{A}$. Para este trabalho definiremos que $\mathbb{A} = 0, 1$ e que todas as funções são completamente especificadas.

Na álgebra booleana, ao contrário da álgebra ordinária dos reais, as variáveis podem assumir apenas um conjunto finito de valores (0 ou 1 neste trabalho). Devido a isso a quantidade de estados de uma função booleana é finito, o que permite que todos possam ser representados. Uma forma comum de representação é através de uma tabela conhecida como **tabela verdade**. Porém, como uma função booleana de n variáveis define um

espaço booleano de 2^n pontos ou estados, temos um complexidade computacional exponencial. Por isso, é necessário encontrar formas eficientes de representar essas funções.

2.3 Expansão de Shannon

Shannon (1949) demonstrou que qualquer função booleana f de n variáveis pode ser escrita da seguinte forma, $f = x_i \cdot f_{x_i} + \overline{x_i} \cdot f_{\overline{x_i}}$, sendo que f_{x_i} e $f_{\overline{x_i}}$ correspondem a f quando o valor da variável x_i é substituído por 1 ou 0 respectivamente. f_{x_i} é chamado de **cofator positivo**, enquanto $f_{\overline{x_i}}$ é chamado de **cofator negativo** de f em relação a x. Apesar de ter ficado conhecido como Teorema da Expansão de Shannon, Boole, o criador da álgebra booleana, já sabia desse princípio.

A aplicação recursiva da expansão de Shannon em uma função irá gerar uma representação única com respeito a ordem das variáveis. A isso se dá o nome de **forma canônica**. Este conceito será importante para subseção 2.4.3 que apresentará o conceito de BDDs ordenados (OBDDs).

2.4 Diagramas de Decisão Binários - BDDs

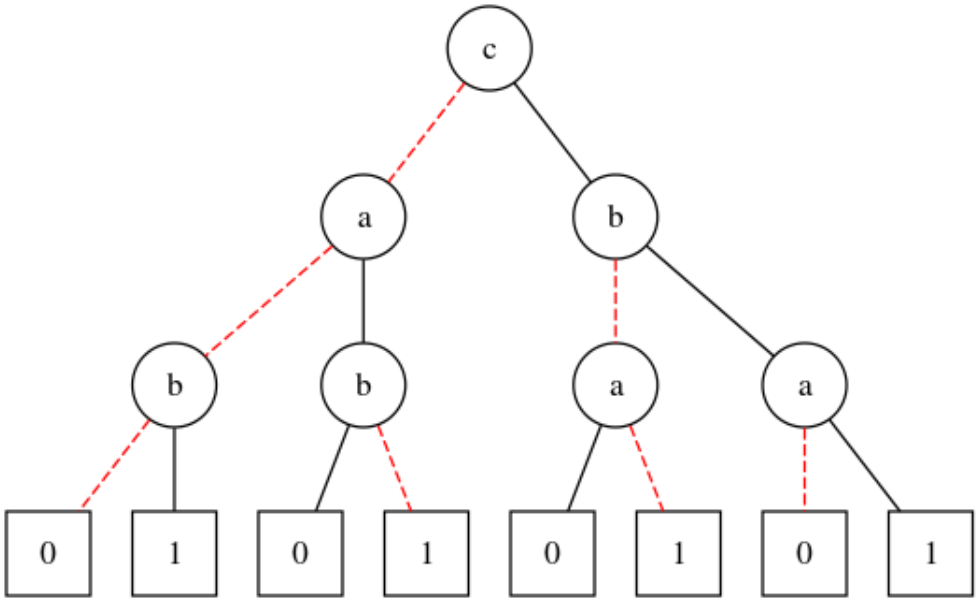
Esta seção descreve os diagramas de decisão binários, os BDDs. Primeiro apresentaremos as árvores de decisão binárias, então apresentaremos tipos de BDDs conforme restrições são aplicadas na árvore de decisão binária: BDDs, OBDDs, ROBDDs (BRYANT, 1986) e ROBDDs na forma fortemente canônica (BRACE; RUDELL; BRYANT, 1990). Os ROBDDs na forma fortemente canônica apresentados na subseção 2.4.5 são o principal interesse deste trabalho. Por uma questão de completude de referências, a subseção 2.4.6 apresenta vários outros tipos de BDDs que foram propostos na literatura.

2.4.1 Árvores de decisão binária

Arvores de decisão binária são árvores obtidas pela aplicação da decomposição de Shannon recursivamente em uma função Booleana. Estas árvores podem ser não ordenadas ou ordenadas. A figura 2.1 apresenta uma árvore não-ordenada, enquanto a figura 2.2 representa uma árvore de decisão binária ordenada.

Note que no caso da árvore de decisão binária não ordenada da figura 2.1, existem

Figura 2.1: Árvore de Decisão Binária (não ordenada) da expressão $a \oplus b \oplus c$



Fonte: Os autores (2019)

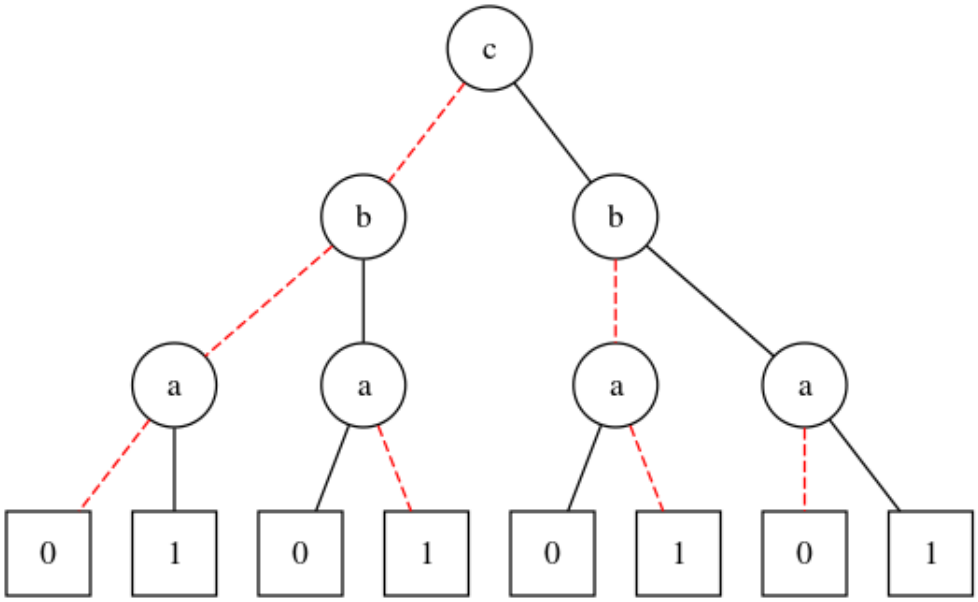
caminhos com a ordem *abc* e com a ordem *cba*. Por outro lado, no caso da árvore de decisão binária ordenada mostrada na figura 2.2, todos os caminhos têm ordem *cba*.

2.4.2 BDDs

BDDs, *binary decision diagrams* ou diagramas de decisão binária, são grafos dirigidos acíclicos que representam funções booleanas. Foram inicialmente propostos por Lee (1959) e depois estudados e difundidos por Akers (1978). Posteriormente, Bryant (1986) aprimorou os BDDs introduzindo os conceito de ordenação e redução, que serão explicados nas subseções 2.4.3 e 2.4.4 respectivamente. Utilizando esses conceitos Brace, Rudell e Bryant (1990) descreveu a implementação de um pacote para manipular BDDs. Janssen (2001) também implementou um pacote de BDDs porém evitando o uso de ponteiros. Minato (2013) apresenta um resumo da história dos BDDs e dos ZDDs, incluindo uma linha do tempo com os trabalhos relacionados.

Cada função booleana possui um nodo inicial(raiz), nodos intermediários e nodos terminais(folhas). Cada nodo de BDD, exceto as folhas, possui as seguintes informações, uma variável v, um filho f_0 que é seu cofator negativo e um filho f_1 que é seu cofator positivo. As folhas podem ter apenas os valores 0 ou 1. A direção do grafo é da raiz até as folhas. A figura 2.3 apresenta um exemplo de BDD. É importante notar que apesar de ser um grafo dirigido, a representação gráfica adota não utiliza setas e sua direção se dá

Figura 2.2: Árvore de Decisão Binária Ordenada da expressão $a \oplus b \oplus c$



Fonte: Os autores (2019)

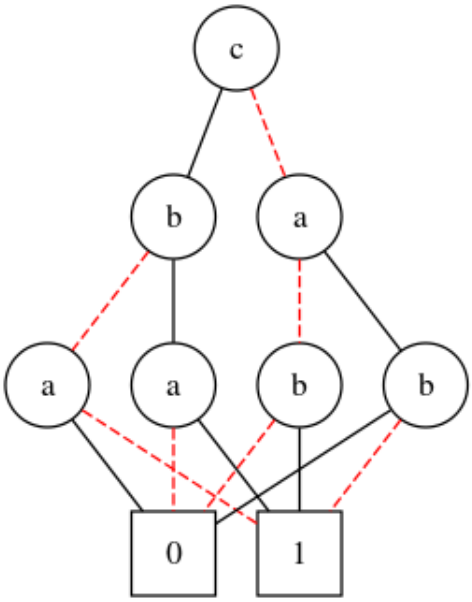
de cima para baixo conforme a ordenação das variáveis. Para diferenciar os cofatores, a aresta do negativo é tracejada, enquanto a aresta do positivo é contínua.

Alguns autores utilizam o termo BDD para se referir aos BDDs ordenados e reduzidos, por serem a forma mais utilizada na prática. Este é o caso de Knuth (2009) em seu livro. Da mesma forma, na maioria das vezes em que o termo é utilizado neste trabalho é para se referir aos ROBDDs.

2.4.3 OBDD

Conforme dito na seção 2.3, através da aplicação da expansão de Shannon completa em uma função booleana obtemos a sua forma em relação à ordem das variáveis em que foi aplicado o teorema. OBDD é a representação em forma de grafo desta expansão. Cada nível i possui somente uma variável x_i associada e cada variável x_i está associada a apenas um nível i, o qu em relação à ordem das variáveis em que foi aplicado o teorema. OBDD é a representação em forma de grafo desta expansão. Cada nível i possui somente uma variável x_i associada e cada variável x_i está associada a apenas um nível i, o que nos permite definir a função bijetora $\pi(i) = x_i$ e sua inversa $\pi^{-1}(x_i) = i$. A ordem das variáveis é importante para a criação dos algoritmos de manipulação dos BDDs. A figura 2.4 apresenta o mesmo exemplo que a figura 2.3 porém após a ordenação.

Figura 2.3: BDD não ordenado da expressão $a \oplus b \oplus c$



Fonte: Os autores (2019)

2.4.4 ROBDD

Apesar de conseguir representar funções booleanas de forma mais eficiente que as tabelas verdades, os BDDs apresentam muitas redundâncias. Por isso, foi criado o conceito de redução, que elimina as redundâncias, através das seguintes transformações:

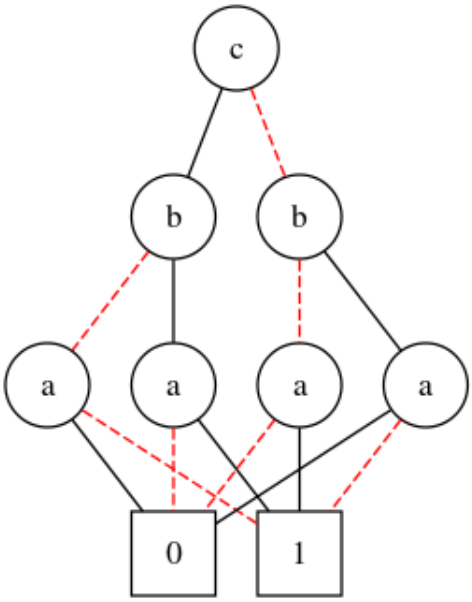
- Elimina as folhas com o mesmo valor deixando somente uma e redireciona as arestas que ligavam as folhas eliminadas para essa única folha. Ou seja, deixa somente uma folha 0 e uma folha 1. A figura 2.5 demonstra essa regra.

- Caso o cofator negativo (f_0) e o cofator positivo (f_1) de um nodo sejam iguais, este nodo é eliminado e as arestas que se dirigiam a ele serão redirecionadas para o f_0. Isso pode ser visto na figura 2.6.

- Caso existem dois nodos com a mesma variável, mesmo cofator positivo e mesmo cofator negativo, então um deles é eliminado e as arestas que se dirigiam a ele serão redirecionadas para o outro nodo. Um exemplo disso está na figura 2.7.

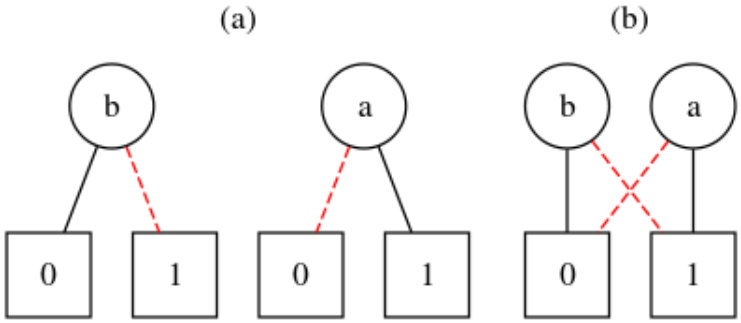
A figura 2.8 apresenta o efeito da redução no exemplo utilizado anteriormente, Pode-se notar que o ROBDD resultante é muito mais compacto e consequentemente mais eficiente que os BDDs das figuras 2.3 e 2.4.

Figura 2.4: OBDD não reduzido da expressão $a \oplus b \oplus c$



Fonte: Os autores (2019)

Figura 2.5: Exemplo de aplicação da primeira regra de redução. (a) antes da aplicação da regra (b) após a aplicação da regra

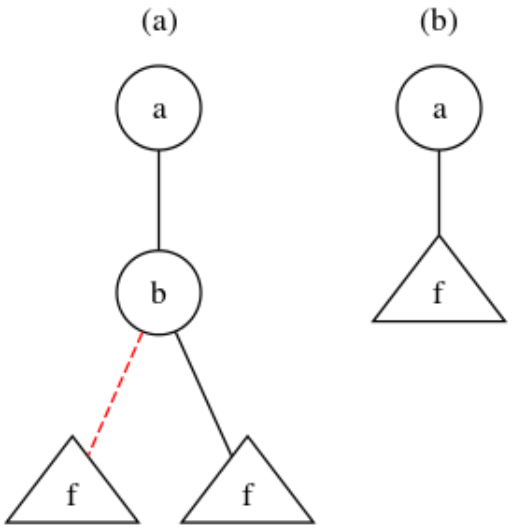


Fonte: Os autores (2019)

2.4.5 Forma fortemente canônica de ROBDDs

Apesar das regras de redução apresentadas na subseção 2.4.4 funcionarem, na prática é muito mais eficiente utilizar uma tabela única que liga uma chave a um nodo de maneira exclusiva. Essa chave pode ser construída através da tupla (v, f_0, f_1). Isso permite testar a existência de um nodo com determinada chave antes de sua criação, evitando assim a criação desnecessária de nodos repetidos. Esta contribuição foi feita por Brace (BRACE; RUDELL; BRYANT, 1990), e é conheccida como forma fortemente canônica dos ROBDDs.

Figura 2.6: Exemplo de aplicação da segunda regra de redução. (a) antes da aplicação da regra (b) após a aplicação da regra



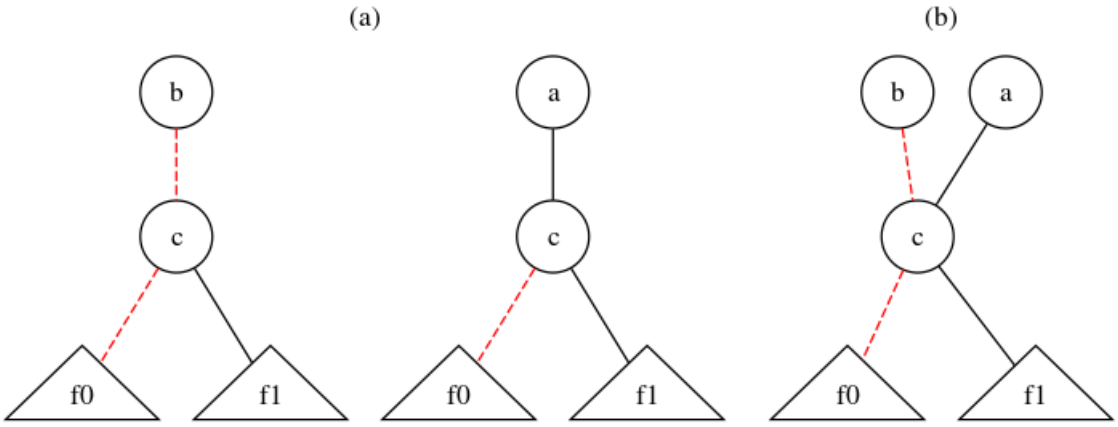
Fonte: Os autores (2019)

2.4.6 Outros tipos de diagramas de decisão

Devido a popularização dos BDDs, muitos outros tipos de diagramas de decisão foram criados e por isso, encontramos uma grande quantidade de acrônimos na literatura (KNUTH, 2009). Colange (2013) menciona e resume alguns destes diagramas, enquanto Knuth (2009) cita outros e trata mais profundamente sobre os ZDDs. Aqui faremos uma breve explicação de alguns diagramas.

IDDs foram criados por Strehl e Thiele (1998) para representar redes de Petri e diferem dos BDDs devido ao conjunto de valores permitidos. Enquanto os BDDs permitem apenas o conjunto 0,1, os IDDs permitem intervalos dos reais. De modo semelhante, os MDDs criados por Srinivasan et al. (1990) diferem dos BDDs por utilizarem subconjuntos dos números naturais, além de permitir que os nodos não-terminais possuam mais de dois filhos. Os MTBDDs introduzidos por Clarke et al. (1993) possuem a característica do domínio de suas funções não ser binário e seu objetivo é calcular a transformada de Walsh. Lai e Sastry (1992) inventou o conceito de EVBDD, no qual as arestas podem assumir valores inteiros para representar equações aritméticas. Minato (1993) sugeriu os ZDDs, que possuem uma regras de redução distintas dos BDDs e são úteis para representar conjuntos de dados esparsos. Dijk, Wille e Meolic (2017) elaborou os TBDDs para utilizar os benefícios dos BDDs e dos ZDDs concomitantemente e para isso criou *tags*

Figura 2.7: Exemplo de aplicação da terceira regra de redução. (a) antes da aplicação da regra (b) após a aplicação da regra



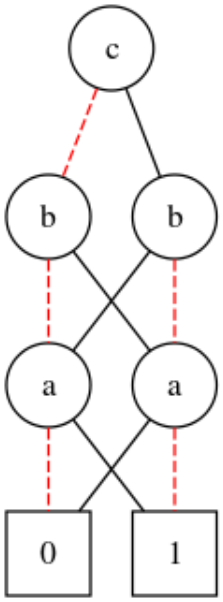
Fonte: Os autores (2019)

para as arestas com o objetivo de indicar qual regra de redução foi utilizada para a remoção de determinada subárvore. Bryant (2018) propôs dois novos tipos de diagramas de decisão, CBDD e CZDD, que são versões de BDD e ZDD que utilizam uma codificação para eliminar cadeias de nodos seguindo terminadas regras.

IBDDs, criados por Jain et al. (1992), são BDDs que possuem mais de uma ordenação de variáveis. Seu objetivo é representar de forma mais eficiente funções que podem ser decompostas em subfunções com ordenação ótima diferente, como por exemplo a multiplicação de n bits. Para essas funções os BDDs são ineficientes, pois nunca haverá uma ordenação ótima. O problema deste método é que como as variáveis podem aparecer mais de uma vez, não existe uma forma canônica.

Além desses tipos de diagramas de decisão, existem muitos outros que foram criados para resolver diversos tipos de problema. Isso acontece, pois esses diagramas demonstraram ser ótimas ferramentas. Na literatura é possível encontrar exemplos do seu uso em navegação de aeronaves, biometria e até decisões de investimentos.

Figura 2.8: ROBDD da expressão $a \oplus b \oplus c$



Fonte: Os autores (2019)

3 BIBLIOTECA BDD BASEADA EM INTEIROS

Neste capítulo serão apresentados os principais aspectos da implementação de uma biblioteca de funções BDD baseadas em inteiros. Essa biblioteca de funções também é, algumas vezes, chamada de pacote e representa um conjunto de funções que permitem executar diversas operações em BDDs. A seção 3.1 trata da codificação de uma chave inteira para a identificação de nodos na tabela única, enquanto a seção 3.2 apresenta o operador ITE que é muito utilizado para implementar outras operações em BDDs. Por fim, a seção 3.3 mostra os nodos resultantes do processamento de uma expressão booleana. Esta biblioteca é baseada na descrita por Gerez (1999).

3.1 Criação da chave inteira para a tabela única

Conforme dito em 2.4.4, a implementação de uma biblioteca BDD eficiente necessita a utilização de uma tabela única que pode ser criada através do uso de uma tabela de dispersão, também conhecida como tabela hash. Com ela podemos aplicar o algoritmo 1 para evitar a construção de nodos repetidos e garantindo assim que cada nodo seja a representação única de uma função, o que é chamado de **forma fortemente canônica**.

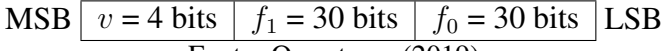
Segundo Liaw e Lin (1992) o quantidade máxima de nodos de uma função booleana de n variáveis é aproximadamente $2^n/n$. Considerando que n é uma potência de dois da forma $n = 2^m$, então o máximo passará a ser 2^{n-m}.

Se para representar um número p em binário são necessários $log_2 p$ bits, então serão necessários m bits para representar n variáveis e $n - m$ bits para indexar cada nodo. Por isso para codificar uma tupla (v, f_0, f_1), a quantidade mínima de bits necessários é $m + 2(n - m)$.

Este trabalho escolheu utilizar números inteiros de 64 bits para codificar a chave da tabela única. Seus campos estão organizados, como pode ser visto na figura 3.1, da seguinte forma: os 4 bits mais significativos representam a variável v, os 30 bits menos significativos identificam o cofator negativo f_0 e os 30 bits restantes são utilizados para indexar o cofator positivo f_1. A utilização de 4 bits para variáveis permite que sejam criadas no máximo 16 delas. En quanto os 30 bits para os cofatores permitem a criação de 2^{30} funções distintas, porém duas delas são criadas previamente, as constantes 0 e 1. Considerando que no pior caso uma função necessita de $2^n/n$ nodos, essa implementação permite a criação de 2^{18} funções com o tamanho máximo.

A alteração do tipo de dado da chave de cadeia de caracteres para inteiro possibilita o ganho de desenho de duas formas, na criação e na comparação. Como internamente as informações armazenadas já eram um número inteiro, então havia uma conversão de inteiro para string na hora de criar a chave, o que não é necessário quando os tipos de dados são iguais. A chave com inteiros pode ser criada utilizando apenas deslocamento de bits, que é uma operação simples. Na comparação cadeira de caracteres são necessárias n operações para obter o resultado final e este n é proporcional ao tamanho da cadeia, porém com inteiros de 64 bits isso é feito sempre em apenas uma operação para arquiteturas de 64 bits ou em duas para arquiteturas de 32 bits. Por outro lado, o uso de inteiros como foi proposto exige um mecanismo para limitar o número de variáveis e de nodos criados, para garantir que um campo não invada outro. Ainda assim, os ganhos obtidos na criação e na comparação da chave compensam as perdas ocasionadas pela mecanismo de limitação.

Figura 3.1: Estrutura da chave única utilizada



Fonte: Os autores (2019)

Algoritmo 1: Novo ou Velho

Entrada: chave t que é uma tupla (v, f_0, f_1)
Saída: o nodo correspondente à tupla

```
início
    se existe um nodo n que corresponde à chave t na tabela única então
        retorna n
    senão
        cria nodo n;
        adiciona nodo n na tabela com a chave t;
        retorna n
```

3.2 Operador ITE

O operador *if-then-else*, conhecido como ITE, é um operador ternário que pode representar qualquer função de duas variáveis (ver tabela 3.1). Ele também pode ser utilizado como base para implementar outras operações nos BDDs (BRACE; RUDELL; BRYANT, 1990). Dadas f, g, h e z funções booleanas, a definição do operador é a seguinte:

$$z = ite(f, g, h) = f \cdot g + \overline{f} \cdot h$$

Este operador é recursivo e sempre retorna um nodo BDD. Olhando sua definição é possível notar uma clara semelhança com a expansão de Shannon mostrada na seção 2.3. O algoritmo 2 apresenta o pseudo-código da sua implementação.

Algoritmo 2: ITE

Entrada: nodos F,G,H e inteiro i

Saída: nodo

início

- **se** $F = t_1$ **então**
  - **retorna** *G*
- **senão se** $F = t_0$ **então**
  - **retorna** *H*
- **senão se** $G = t_1$ *e* $H = t_0$ **então**
  - **retorna** *F*
- **senão**
  - $v \leftarrow \pi(i)$;
  - $f_0 \leftarrow ite(F_v, G_v, H_v, i+1)$;
  - $f_1 \leftarrow ite(F_{\overline{v}}, G_{\overline{v}}, H_{\overline{v}}, i+1)$;
  - **se** f_0 = f_1 **então**
    - **retorna** f_0
  - **senão**
    - **retorna** $NovoOuVelho(v.f_0, f_1)$

Tabela 3.1: Equivalência do operador ITE com operações de duas entradas

Nome	Expressão	ITE
f	f	ite(f,1,0)
$NOT(f)$	$\overline{f}$	ite(f,0,1)
$AND(f,g)$	$f \cdot g$	ite(f,g,0)
$OR(f,g)$	$f + g$	ite(f,1,g)
$XOR(f,g)$	$f \oplus g$	ite(f,$\overline{g}$,g)
$XNOR(f,g)$	$f \otimes g$	ite(f,g,$\overline{g}$)
$NAND(f,g)$	$\overline{f \cdot g}$	ite(f,$\overline{g}$,1)
$NOR(f,g)$	$\overline{f + g}$	ite(f,0,$\overline{g}$)

Fonte: Adaptada de Brace, Rudell e Bryant (1990)

3.3 Exemplo de execução

A tabela 3.2 apresenta os nodos BDD que foram criados durante o processamento da função *XNOR* de três entradas. Os nodos t0 e t1 foram criados previamente para representar as folhas 0 e 1. Devido a aplicação sucessiva do operador ITE, são criados nodos que representam subfunções da função escolhida. Por exemplo, os nodos n1, n2 e n5 são os literais positivos das variáveis a, b e c, enquanto os nodos n3 e n6 são os literais negativos das variáveis b e c. Cada um desses nodos que representam literais pode ser encarado como uma função de uma única entrada. Posteriormente esses nodos são combinados para compor outras funções através da utilização de operadores lógicos. O nodo n10 é a raiz do BDD que representa $a \otimes b \otimes c$ e é obtido como o complementar do nodo n9.

Tabela 3.2: Nodos criados pela execução do algoritmo para representar a expressão $a \otimes b \otimes c$

nodo	v	f_0	f_1	expressão
t0	-	-	-	0
t1	-	-	-	1
n1	a	t0	t1	a
n2	b	t0	t1	b
n3	b	t1	t0	$\overline{b}$
n4	a	n2	n3	$\overline{a}b + a\overline{b}$
n5	c	t0	t1	c
n6	c	t1	t0	$\overline{c}$
n7	b	n5	n6	$\overline{b}c + b\overline{c}$
n8	b	n6	n5	$\overline{bc} + bc$
n9	a	n7	n8	$\overline{a\overline{b}c} + \overline{a}b\overline{c} + a\overline{bc} + abc$
n10	a	n8	n7	$\overline{abc} + \overline{a}bc + a\overline{b}c + ab\overline{c}$

Fonte: Os autores (2019)

4 RESULTADOS EXPERIMENTAIS

Neste capítulo descreveremos a metodologia utilizada para a realização dos experimentos. Os resultados obtidos serão comparados com os da implementação de referência. Com isso, podemos avaliar a utilidade da imprementação proposta no capítulo 3 deste trabalho.

4.1 Metologia

Este trabalho implementou uma biblioteca de funções que utiliza ROBDDs na forma fortemente canônica com a chave única baseada em inteiros utilizando a linguagem C++ conforme descrito no capítulo 3. Esta implementação foi comparada com uma implementação de ROBDDs, também na forma fortemente canônica, utilizada na cadeira de *CAD para Sistemas Digitais*, cuja chave única é baseada em cadeia de caracteres. Ambas as implementações foram compiladas pelo gcc 5.4.0 no sistema operacional Ubuntu 16.04.6 LTS rodando num notebook cujo processador é um Intel Core i5-4200U que possui arquitetura 64 bits.

Para comparação, foi utilizado o seguinte conjunto de funções booleanas, *and*, *or*, *xor*, *nand*, *nor* e *xnor*. Para cada uma destas funções, o número de entradas foi de duas a dezesseis. Além destas funções, também foram implementadas funções que representam os seguintes circuitos aritméticos: somadores, subtratores e multiplicadores. Para os circuitos aritméticos foram usadas duas entradas de 2 a 8 bits (de modo a ter 16 entradas no total).

As implementações dos benchmarks podem ser vistas em pseudocódigo nos algoritmos 3, 4 e 5. O tempo medido é o de criação dos nodos que representam às saídas dos circuitos correspondentes. Para as portas lógicas, o teste consistiu em utilizar uma função da biblioteca de BDDs que transforma uma expressão escrita como string em um BDD, retornando o nodo que corresponde a sua raiz. O teste de somadores e subtratores seguiu o mesmo princípio, porém como possuem múltiplas saídas, foi necessário o uso de múltiplas expressões. Como os multiplicadores são mais complexos, as expressões criadas são muito grandes, então a abordagem foi diferente. Foram utilizadas as funções da biblioteca que criam variáveis e executam operações lógicas baseadas em ITE. No algoritmo 5, a função *ProdutoVetorialBooleano* corresponde ao produto vetorial das entradas a e b, que são vetores de nodos de tamanho n, em que a multiplicação é substituída pelo *AND*

lógico e a saída é uma matrix ab de n por n em que cada elemento é também é um nodo. Ou seja, cada elemento ab_{ij} corresponde a $AND(a_i, bj)$. Em seguida esses resultados são "somados"como num *Array Multiplier*, o que é traduzido como operações *XOR* e *AND* entre nodos, de forma a equivaler às saídas de um somador.

O desempenho das implementações foi comparado através do tempo de execução médio de cem execuções consecutivas. Este método visa diminuir a influência de tempos de execução anormais que ocorrem devido a utilização do computador e o escalonamento dos processos.

Algoritmo 3: Benchmark de Portas Lógicas

```
início
    tinicio = TempoAtual();
    nodo = CriaBddDaEquação(equação, bdd);
    tfim = TempoAtual();
    tduração = tfim- tinicio;
    memória = PicoUsoMemória();
    Imprime(tduração, memória, bdd.qtdNodos);
```

Algoritmo 4: Benchmark de Somadores e Subtratores

```
início
    tinicio = TempoAtual();
    para todo saídaDoCircuito faça
        nodo = CriaBddDaEquação(saídaDoCircuito, bdd);
    tfim = TempoAtual();
    tduração = tfim- tinicio;
    memória = PicoUsoMemória();
    Imprime(tduração, memória, bdd.qtdNodos);
```

4.2 Impacto do utilização da chave inteira

As tabelas abaixo apresentam uma comparação dos tempos obtidos na criação dos BDDs que representam o conjunto de funções utilizado. As colunas possuem a seguinte configuração, n é a quantidade de entradas (no caso dos somadores e subtratores as entradas são dois vetores de n bits), q é a quantidade de nodos não-terminais criados, t_i é o tempo de execução do pacote baseado em inteiros, t_s é o tempo de execução do pacote baseado em strings e t_s/t_i é a razão entre os tempos obtidos por cada um dos pacotes.

Algoritmo 5: Benchmark de Multiplicadores

```
início
    tinicio = TempoAtual();
    CriaVariáveis(a);
    CriaVariáveis(b);
    ProdutoVetorialBooleano(a,b,ab);
    SomasDoMultiplicador(ab,p);
    tfim = TempoAtual();
    tduração = tfim- tinicio;
    memória = PicoUsoMemória();
    Imprime(tduração, memória, bdd.qtdNodos);
```

Também foram efetuadas medidas da quantidade de memória máxima utilizada durante a execução do programa, porém a diferença entre as implementações na maior parte dos benchmarks usados não foi relevante e por isso seus dados foram omitidos. O único que apresentou variações em alguns caso foi o circuito multiplicador e seus resultados podem ser vistos na tabela 4.9.

Tabela 4.1: Comparação dos tempos de execução da função and

n	q	$t_i(\mu s)$	$t_s(\mu s)$	t_s / t_i
2	3	22,12	30,82	1,39
3	6	22,69	39,34	1,73
4	10	31,79	41,58	1,31
5	15	27,45	45,81	1,67
6	21	30,77	56,63	1,84
7	28	33,44	63,01	1,88
8	36	35,01	74,18	2,12
9	45	45,29	91,29	2,02
10	55	47,79	99,58	2,08
11	66	47,28	110,17	2,33
12	78	52,87	125,25	2,37
13	91	57,12	137,79	2,41
14	105	62,22	164,42	2,64
15	120	67,68	179,61	2,65
16	136	76,51	196,93	2,57

Fonte: Os autores (2019)

Como pode ser visto na tabela 4.1, os tempos de execução obtidos foram até 2.57

vezes menores para a implementação com a chave única baseada em inteiros. É importante perceber que a vantagem da implementação com inteiros é maior para funções com um maior número de entradas.

Tabela 4.2: Comparação dos tempos de execução da função *or*

n	q	$t_i(\mu s)$	$t_s(\mu s)$	t_s / t_i
2	3	24,67	30,10	1,22
3	6	24,37	35,81	1,47
4	10	26,69	40,60	1,52
5	15	30,57	47,56	1,56
6	21	30,21	53,56	1,77
7	28	35,60	64,04	1,80
8	36	36,39	73,05	2,01
9	45	46,65	85,26	1,83
10	55	43,96	97,51	2,22
11	66	47,16	111,24	2,36
12	78	49,97	123,62	2,47
13	91	61,63	136,19	2,21
14	105	64,66	156,83	2,43
15	120	67,41	179,25	2,66
16	136	74,50	195,47	2,62

Fonte: Os autores (2019)

Como pode ser visto na tabela 4.2, os resultados são similares aos obtidos para a função and (4.1). Os tempos de execução obtidos foram até 2.62 vezes menores para a implementação com a chave única baseada em inteiros. É importante perceber que novamente a vantagem da implementação com inteiros é maior para funções com um maior número de entradas.

Tabela 4.3: Comparação dos tempos de execução da função xor

n	q	$t_i(\mu s)$	$t_s(\mu s)$	t_s / t_i
2	4	21,47	32,18	1,50
3	9	26,21	41,57	1,59
4	16	28,87	54,93	1,90
5	25	31,09	71,15	2,29
6	36	36,95	102,60	2,78
7	49	45,85	159,05	3,47
8	64	60,18	269,47	4,48
9	81	75,60	481,33	6,37
10	100	114,03	888,55	7,79
11	121	202,65	1755,65	8,66
12	144	313,43	3344,80	10,67
13	169	619,68	6648,73	10,73
14	196	1034,86	13074,40	12,63
15	225	2240,15	25801,92	11,52
16	256	3898,51	51765,98	13,28

Fonte: Os autores (2019)

Já para a função xor, mostrada na tabela 4.3, os tempos de execução obtidos foram até 13.28 vezes menores para a implementação com a chave única baseada em inteiros. É importante perceber que a vantagem da implementação com inteiros é maior para funções com um maior número de entradas. Aqui o ganho é maior também porque a xor tem uma complexidade maior comparada às demais. Ou seja, o ganho obtido pela implementação da chave única como um único inteiro parece ser maior para funções mais complexas.

Tabela 4.4: Comparação dos tempos de execução da função $nand$

n	q	$t_i(\mu s)$	$t_s(\mu s)$	t_s / t_i
2	5	25,09	34,50	1,38
3	9	25,23	40,43	1,60
4	14	29,30	45,37	1,55
5	20	34,03	54,41	1,60
6	27	32,25	63,06	1,96
7	35	36,79	73,92	2,01
8	44	41,91	84,92	2,03
9	54	43,50	98,24	2,26
10	65	47,25	109,93	2,33
11	77	50,47	123,65	2,45
12	90	59,17	138,58	2,34
13	104	60,28	159,43	2,64
14	119	64,16	174,50	2,72
15	135	71,11	197,27	2,77
16	152	78,76	216,97	2,75

Fonte: Os autores (2019)

No caso da função nand, cujos dados de execução são mostrados na tabela 4.4, os tempos de execução obtidos foram até 2.75 vezes menores para a implementação com a chave única baseada em inteiros. É importante perceber que a vantagem da implementação com inteiros é maior para funções com um maior número de entradas.

Tabela 4.5: Comparação dos tempos de execução da função nor

n	q	$t_i(\mu s)$	$t_s(\mu s)$	t_s / t_i
2	5	22,97	32,32	1,41
3	9	25,56	40,41	1,58
4	14	27,60	50,49	1,83
5	20	28,57	54,44	1,91
6	27	32,47	63,08	1,94
7	35	38,41	75,48	1,97
8	44	40,23	85,76	2,13
9	54	44,88	103,51	2,31
10	65	46,88	107,98	2,30
11	77	48,51	131,51	2,71
12	90	57,84	138,13	2,39
13	104	65,09	156,14	2,40
14	119	67,19	173,11	2,58
15	135	77,14	195,69	2,54
16	152	78,12	215,43	2,76

Fonte: Os autores (2019)

A função nor, mostrada na tabela 4.5, tem um comportamento muito próximo a função nand. Os tempos de execução obtidos foram até 2.76 vezes menores para a implementação com a chave única baseada em inteiros. É importante perceber que a vantagem da implementação com inteiros é maior para funções com um maior número de entradas.

Tabela 4.6: Comparação dos tempos de execução da função $xnor$

n	q	$t_i(\mu s)$	$t_s(\mu s)$	t_s / t_i
2	5	23,08	37,56	1,63
3	10	24,56	48,61	1,98
4	17	30,45	65,02	2,14
5	26	33,78	95,89	2,84
6	37	42,74	157,69	3,69
7	50	55,24	265,17	4,80
8	65	76,16	471,42	6,19
9	82	132,82	889,59	6,70
10	101	179,22	1680,78	9,38
11	122	315,50	3378,80	10,71
12	145	562,39	6546,72	11,64
13	170	1103,47	13077,40	11,85
14	197	2030,54	25852,22	12,73
15	226	4208,81	51679,79	12,28
16	257	7777,33	103570,08	13,32

Fonte: Os autores (2019)

A função xnor, mostrada na tabela 4.6, tem um comportamento muito próximo a função xor. Os tempos de execução obtidos foram até 13.32 vezes menores para a implementação com a chave única baseada em inteiros. É importante perceber que a vantagem da implementação com inteiros é maior para funções com um maior número de entradas.

Tabela 4.7: Comparação dos tempos de execução dos somadores

n	q	$t_i(\mu s)$	$t_s(\mu s)$	t_s / t_i
2	19	44,12	84,06	1,91
3	45	69,75	182,56	2,62
4	83	118,07	426,54	3,61
5	133	208,80	1064,08	5,10
6	195	416,33	2830,97	6,80
7	269	1017,38	7897,52	7,76
8	355	2599,88	23117,34	8,89

Fonte: Os autores (2019)

No caso dos somadores, mostrados na tabela 4.7, os tempos de execução obtidos

foram até 8.89 vezes menores para a implementação com a chave única baseada em inteiros. É importante perceber que a vantagem da implementação com inteiros é maior para funções com um maior número de entradas.

Tabela 4.8: Comparação dos tempos de execução dos subtratores

n	q	$t_i(\mu s)$	$t_s(\mu s)$	t_s / t_i
2	21	45,90	91,04	1,98
3	55	81,84	208,12	2,54
4	94	124,77	511,77	4,10
5	145	216,29	1200,22	5,55
6	208	419,30	3043,02	7,26
7	283	1033,25	8230,20	7,97
8	370	2498,17	23479,84	9,40

Fonte: Os autores (2019)

Os subtratores, mostrados na tabela 4.8, tem um comportamento similar ao dos somadores. No caso dos subtratores, os tempos de execução obtidos foram até 9.40 vezes menores para a implementação com a chave única baseada em inteiros. É importante perceber que a vantagem da implementação com inteiros é maior para funções com um maior número de entradas.

Tabela 4.9: Comparação dos tempos de execução dos multiplicadores

n	q	$t_i(\mu s)$	$t_s(\mu s)$	t_s / t_i	$m_i(MB)$	$m_s(MB)$	m_s / m_i
2	20	38,29	80,57	2,10	2,38	2,35	0,99
3	125	101,17	351,85	3,48	2,37	2,35	0,99
4	560	381,00	2015,04	5,29	2,42	4,03	1,67
5	2081	1608,89	9555,48	5,94	4,14	4,15	1,00
6	7230	7537,39	46163,41	6,12	4,66	4,66	1,00
7	24410	35713,81	220048,72	6,16	6,24	6,97	1,12
8	79840	166021,95	1047389,65	6,31	11,46	14,77	1,29

Fonte: Os autores (2019)

No caso dos multiplicadores, mostrados na tabela 4.9, os tempos de execução obtidos foram até 6.31 vezes menores para a implementação com a chave única baseada em inteiros. É importante perceber que a vantagem da implementação com inteiros é maior para funções com um maior número de entradas.

Considerando uma análise geralde todas as funções apresentadas nas tabelas anteriores, os resultados demonstraram os benefícios da utilização da condificação em nú-

meros inteiros da chave hash. O seu uso permitiu diminuir o tempo de execução para processar as diversas expressões booleanas, além de manter os mesmos resultados.

As seguintes conclusões puderam ser obtidas através das tabelas:

- O tempo de execução da implementação deste trabalho foi menor que a implementação baseada em strings em todos os casos.
- Quanto maior a quantidade de entradas, maior a diferença de desempenho entre os códigos.
- As funções XOR e XNOR foram as que obtiveram o maior aumento de velocidade. O tempo de execução destas funções no pacote bdd baseado em strings chegou a ser mais de 1200% maior que o obtido no pacote proposto neste trabalho.

5 CONCLUSÃO

Este trabalho de conclusão apresentou uma implementação mais eficiente de um pacote de BDDs. A eficiência foi obtida pela compactação da chave única de ROBDDs na forma fortemente canônica em um único inteiro. Esta eficiência foi obtida com algumas penalidades, a principal penalidade é a representação de funções menores, já que os três elementos da chave única devem ser armazenados em um único inteiro. Porém, em termos de tempo de execução foi obtida uma implementação que pode ser uma ordem de grandeza mais rápida quando comparada com a implementação de referência que usava cadeias de caracteres para formar a chave única. Em conclusão, foi obtida uma implementação de ROBDDs na forma fortemente canônica que é altamente eficiênte para representação e manipulação de pequenas funções Booleanas até 16 entradas. Este pacote pode ser útil em um contexto de otimização de circuitos maiores divididos em cortes K ou cortes KL restritos a um máximo de 16 entradas (K=16). Este é um contexto bem realista dentro de um escopo de síntese lógica.

REFERÊNCIAS

AKERS, S. B. Binary decision diagrams. **IEEE Transactions on computers**, IEEE, n. 6, p. 509–516, 1978.

BRACE, K. S.; RUDELL, R. L.; BRYANT, R. E. Efficient implementation of a bdd package. In: **Proceedings of the 27th ACM/IEEE Design Automation Conference**. New York, NY, USA: ACM, 1990. (DAC '90), p. 40–45. ISBN 0-89791-363-9. Disponível em: <http://doi.acm.org/10.1145/123186.123222>.

BRYANT, R. E. Graph-based algorithms for boolean function manipulation. **IEEE Transactions on Computers**, C-35, n. 8, p. 677–691, Aug 1986. ISSN 0018-9340.

BRYANT, R. E. Chain reduction for binary and zero-suppressed decision diagrams. In: SPRINGER. **International Conference on Tools and Algorithms for the Construction and Analysis of Systems**. [S.l.], 2018. p. 81–98.

BUTZEN, P. F. et al. Standby power consumption estimation by interacting leakage current mechanisms in nanoscaled cmos digital circuits. **Microelectronics Journal**, Elsevier, v. 41, n. 4, p. 247–255, 2010.

CLARKE, E. M. et al. Spectral transforms for large boolean functions with applications to technology mapping. In: IEEE. **30th ACM/IEEE Design Automation Conference**. [S.l.], 1993. p. 54–60.

COLANGE, M. Symmetry reduction and symbolic data structures for model checking of distributed systems. **Context**, v. 2, p. 29, 2013.

DIJK, T. van; WILLE, R.; MEOLIC, R. Tagged bdds: combining reduction rules from different decision diagram types. In: FMCAD INC. **Proceedings of the 17th Conference on Formal Methods in Computer-Aided Design**. [S.l.], 2017. p. 108–115.

GEREZ, S. H. **Algorithms for VLSI Design Automation**. 1st. ed. New York, NY, USA: John Wiley & Sons, Inc., 1999. ISBN 0471984892.

HUNTINGTON, E. V. Sets of independent postulates for the algebra of logic. **Transactions of the American Mathematical Society**, American Mathematical Society, v. 5, n. 3, p. 288–309, 1904. ISSN 00029947. Disponível em: <http://www.jstor.org/stable/1986459>.

JAIN, J. et al. Ibdds: An efficient functional representation for digital circuits. In: IEEE. **[1992] Proceedings The European Conference on Design Automation**. [S.l.], 1992. p. 440–446.

JANSSEN, G. Design of a pointerless bdd package. In: **Proceedings of the 10th Int. Workshop on Logic and Synthesis**. [S.l.: s.n.], 2001. (IWLS 2001), p. 310–315.

JUNIOR, L. S. da R. et al. Fast disjoint transistor networks from bdds. In: ACM. **Proceedings of the 19th annual symposium on Integrated circuits and systems design**. [S.l.], 2006. p. 137–142.

KNUTH, D. **The art of computer programming**. Boston, Mass. London: Addison-Wesley, 2009. ISBN 9780321637413.

LAI, Y.-T.; SASTRY, S. Edge-valued binary decision for multi-level hierarchical verification. In: IEEE. **[1992] Proceedings 29th ACM/IEEE Design Automation Conference**. [S.l.], 1992. p. 608–613.

LEE, C. Y. Representation of switching circuits by binary-decision programs. **The Bell System Technical Journal**, v. 38, n. 4, p. 985–999, July 1959. ISSN 0005-8580.

LIAW, H. T.; LIN, C. S. On the obdd-representation of general boolean functions. **IEEE Transactions on Computers**, v. 41, n. 6, p. 661–664, June 1992. ISSN 0018-9340.

MARTINS, M. G. et al. Boolean factoring with multi-objective goals. In: IEEE. **Computer Design (ICCD), 2010 IEEE International Conference on**. [S.l.], 2010. p. 229–234.

MARTINS, M. G.; RIBAS, R. P.; REIS, A. I. Functional composition: A new paradigm for performing logic synthesis. In: IEEE. **Quality Electronic Design (ISQED), 2012 13th International Symposium on**. [S.l.], 2012. p. 236–242.

MINATO, S.-i. Techniques of bdd/zdd: brief history and recent activity. **IEICE TRANSACTIONS on Information and Systems**, The Institute of Electronics, Information and Communication Engineers, v. 96, n. 7, p. 1419–1429, 2013.

MINATO, S. ichi. Zero-suppressed bdds for set manipulation in combinatorial problems. **30th ACM/IEEE Design Automation Conference**, p. 272–277, 1993.

MISHCHENKO, A. et al. Abc: A system for sequential synthesis and verification. **URL http://www. eecs. berkeley. edu/alanmi/abc**, v. 17, 2007.

MOORE, G. E. et al. **Cramming more components onto integrated circuits**. [S.l.]: McGraw-Hill New York, NY, USA:, 1965.

MOREIRA, M. et al. Semi-custom ncl design with commercial eda frameworks: Is it possible? In: **International Symposium on Asynchronous Circuits and Systems (ASYNC), Potsdam**. [S.l.: s.n.], 2014.

ROSA, L. et al. Scheduling policy costs on a java microcontroller. In: SPRINGER. **On The Move to Meaningful Internet Systems 2003: OTM 2003 Workshops**. [S.l.], 2003. p. 520–533.

SHANNON, C. E. The synthesis of two-terminal switching circuits. **The Bell System Technical Journal**, v. 28, n. 1, p. 59–98, Jan 1949. ISSN 0005-8580.

SOEKEN, M. et al. **The EPFL Logic Synthesis Libraries**. 2018.

SRINIVASAN, A. et al. Algorithms for discrete function manipulation. In: **1990 IEEE International Conference on Computer-Aided Design. Digest of Technical Papers**. [S.l.: s.n.], 1990. p. 92–95.

STREHL, K.; THIELE, L. Symbolic model checking using interval diagram techniques. **TIK-Report**, Computer Engineering and Networks Laboratory (TIK), Swiss Federal Institute . . . , v. 40, 1998.

TOGNI, J. et al. Automatic generation of digital cell libraries. In: IEEE. **Proceedings. 15th Symposium on Integrated Circuits and Systems Design**. [S.l.], 2002. p. 265–270.

WILTGEN, A. et al. Power consumption analysis in static cmos gates. In: IEEE. **2013 26th Symposium on Integrated Circuits and Systems Design (SBCCI)**. [S.l.], 2013. p. 1–6.